Java X. – Altípusos polimorfizmus

# Az altípus:

Ha A-t B-ből származtatjuk, akkor A, B egy altípusa

Például public class ExactTime extends Time { ... } esetén:

## Liskov-féle helyettesítési elv (LSP):

Legyen A, a B bázistípus egy altípusa, ekkor A egyedivel helyettesíthetjük B egyedeit anélkül, hogy ez hibás működést eredményezne.

# Polimorf referenciák:

ExactTime time1 = new ExactTime();

Time time2 = new ExactTime(); // upcast

time2.sameHourAs( time1 )

A második sorban a LSP értelmében az értékadás jobb oldalán az általánosabb Time osztályt helyettesíthetjük a specifikusabb ExactTime osztállyal.

Statikus/deklarált típusnak nevezzük egy változó azon típusát, mely a programszövegből következik, amit a fordítóprogram ellenőriz.

Time time

Dinamikus/tényleges típusnak nevezzük, azt a típust, (mely a változó statikus típusának egy altípusa,) mellyel a változó ténylegesen futási időben jelentkezik

time = ... ? new ExactTime() : new Time()

# Dinamikus kötés (késői kötés):

Ha meghívunk egy objektumra egy műveletet akkor mindig az arra leginkább illő implementáció fut le. Ez a „legspeciálisabb” megvalósítás futási időben kerül kiválasztásra.

ExactTime e = new ExactTime();

Time t = e;

Object o = t;

System.out.println( e.toString() ); // 0:00:00

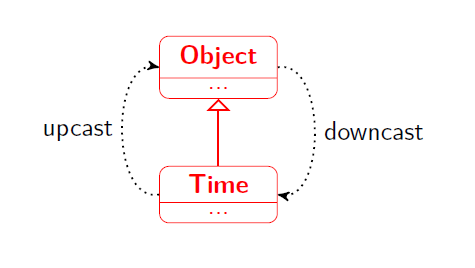
System.out.println( t.toString() ); // 0:00:00

System.out.println( o.toString() ); // 0:00:00

# Dinamikus típusellenőrzés:

Upcast automatikus

Downcast manuális a type-cast operátorral



Ha o dinamikus típusa Time, azért *downcastolhatjuk* Time-á:

Object o = new Time(3,20);

o.aMinutePassed(); // fordítási hiba

((Time)o).aMinutePassed(); // lefordul, működik

Ha mégsem Time a dinamikus típus akkor ClassCastException lép fel.

Object o = "Három óra húsz";

((Time)o).aMinutePassed(); // típusellenőrzés -> futási hiba

A dinamikus típusellenőrzés futási időben történik, a dinamikus típus alapján ezáltal pontosabb, mint a statikus típusellenőrzés.

A Java biztosít egy speciális operátort az instanceof-t, mellyel ellenőrizhetjük megfelelő-e (altípusa-e a megadott típusnak) a dinamikus típus.

Object o = new ExactTime(3,20,0);

...

if( o instanceof Time ){

((Time)o).aMinutePassed();

}

A JVM a program futása során minden objektum dinamikus típusát nyilvántartja. Ezt úgy oldja meg hogy minden használt osztályhoz tartozik egy azt reprezentáló java.lang.Class típusú objektum.

Object o = new Time(17,25);

Class c = o.getClass(); // Time.class

Class cc = c.getClass(); // Class.class

A getClass() metódussal futási időben lekérhető az adott objektum osztálya.

# Típuskonverziók:

## Primitív típusok:

### Automatikus konverziók (tranzitív):

* (veszteséges)

byte b = 42; és short s = 42; és char c = 42;

### Explicit kikényszerítés (type cast):

int i = 42;

short s = (short)i;

### Egy érdekes példa (forrás: Java Puzzlers):

public class LongDivision {

public static void main(String[] args) {

final long MICROS\_PER\_DAY = 24 \* 60 \* 60 \* 1000 \* 1000;

final long MILLIS\_PER\_DAY = 24 \* 60 \* 60 \* 1000;

System.out.println(MICROS\_PER\_DAY / MILLIS\_PER\_DAY);

}

}

A második 1000-rel való szorzás hatására a szorzat túlcsordul majd ez a túlcsordult int kerül konvertálásra. A program könnyen javítható, ha valamelyik tényezőről megmondjuk, hogy long literál (pl. 1000l).

## Primitív típusok és referenciák:

Mutálhatatlan csomagoló osztályok (pl. java.lang.Integer)

Automatikus ki- és becsomagolás (Auto- (un)boxing). Ez a két kódrészlet ekvivalens.

ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(7);

int seven = numbers.get(0);

ArrayList<Integer> numbers = new ArrayList<>();

numbers.add( Integer.valueOf(7) );

int seven = numbers.get(0).intValue();